Mise en œuvre d'un réseau expérimental de capteurs sans fil et application domotique

Anthony Deroche, Thierry Duhal

Projets interdisciplinaires ou de découverte de la recherche

Mise en oeuvre d'un réseau expérimental de capteurs sans-fil et application domotique

13 mai 2014

Deroche Anthony, Duhal Thierry

Table des matières

1 Introduction

1.1 Environnement de travail

Le Projet Interdisciplinaire ou de Découverte de la Recherche (PIDR) a pour objectif de faire découvrir le monde de la recherche aux étudiants de TELECOM Nancy. Ce projet s'avère être de plus grande envergure que ceux habituellement réalisés dans le cadre de notre formation, dans la mesure où il s'étale sur une durée de 6 mois, commençant début janvier et se terminant fin mai 2014.

Nous avons choisi d'effectuer notre PIDR au sein du Laboratoire Lorrain de Recherche en Informatique et ses Applications (LORIA). Ce dernier regroupe un grand nombre d'équipes de recherche pouvant être composées de chercheurs, d'enseignants-chercheurs, d'ingénieurs, de doctorants ou encore de post doctorants. Les thématiques de recherches sont variées, bien qu'ayant un lien avec l'informatique.

L'équipe de recherche dans laquelle nous avons réalisé notre projet se prénomme MADYNES (MAnaging DYnamic Networks and Services). Cette équipe est principalement spécialisée dans le domaine des réseaux informatiques. Elle s'intéresse notamment à l'étude de nouvelles approches pour configurer, maintenir et sécuriser les protocoles de communications au niveau de l'Internet futur (l'Internet des objets).

Au cours de ce projet, nous avons été encadrés par M. Thibault CHOLEZ et M. Emmanuel NATAF, que nous tenons à remercier pour leur accueil, leur disponibilité, ainsi que pour les précieux conseils qu'ils nous ont fournis tout au long de ce semestre. Nous tenons également à remercier M. Arthur Garnier, étudiant à l'IUT Charlemagne de Nancy et stagiaire au LORIA pendant notre projet, qui a contribué au développement de certaines parties de l'application.

1.2 Contexte de recherche

De nos jours, la plupart des objets qui nous entourent ont la vocation d'être connectés. Étant équipés d'une puce ou d'un capteur, ils se mettent ainsi à produire des données. Dans ce contexte de "l'Internet des objets", l'innovation est importante et c'est ainsi que de nombreuses applications voient le jour. On peut par exemple citer les domaines d'applications tels que la planification urbaine (environnement urbain durable, éclairage public), la gestion des déchets, la domotique (gestion du chauffage et de l'éclairage, alarmes de sécurité). Ces nouvelles applications font souvent appel à des réseaux de capteurs sans-fil, ces derniers faisant l'objet d'un travail de recherche important, notamment en termes de sécurité et d'optimisation. C'est dans ce contexte de recherche que s'inscrit notre projet.

1.3 Objectifs poursuivis

Dans le cadre de ce PIDR, nous avons à notre disposition des capteurs permettant de mesurer la température, l'humidité et la luminosité. Notre objectif est de déployer ce réseau de capteurs sans-fil au sein de TELECOM Nancy, afin d'étudier les limites des capteurs actuels. Les données ainsi collectées sont centralisées dans une base de données, rendues publiques, et libres d'être utilisées par diverses applications. Cela s'inscrit bien dans la tendance actuelle des données ouvertes (Open Data).

Afin de collecter ces données, nous avons mis en oeuvre toute une architecture qui sera détaillée par la suite. Nous vous présenterons tout d'abord le matériel et les outils déjà existants mis à notre disposition, avant de vous détailler les choix qui nous ont amenés à réaliser une nouvelle application de collecte des données. Par la suite, nous vous expliquerons de manière relativement précise le fonctionnement de notre application. Enfin, nous analyserons le résultat du déploiement du réseau de capteurs au sein de TELECOM Nancy, et nous discuterons des éventuelles améliorations que nous envisageons.

2 Organisation et moyens mis en oeuvre

2.1 Présentation de notre base de travail

Cette partie détaille le contexte dans lequel nous avons commencé ce projet, à savoir le matériel de base et les outils dont nous disposions, l'organisation de notre travail, ainsi que les choix que nous avons fait au départ.

2.1.1 Matériel à notre disposition

Lorsque nous avons commencé ce PIDR, nos encadrants nous ont fournis quelques-uns des « motes » nécessaires à la réalisation de ce projet. Par « mote » nous entendons un « noeud capteur » du réseau, que voici ci-dessous.

Figure 2.1: Photo d'un noeud du réseau

Conçus par l'université de Californie à Berkeley, ce mote est de type TelosB et se prénomme MTM-CM5000-MSP. Il est composé d'un microcontrôleur MSP430, d'un émetteur-récepteur, d'un capteur de température et d'humidité, d'un capteur de luminosité pour le domaine du visible, d'un autre pour l'infrarouge, d'un « bouton utilisateur », d'un bouton reset, de trois LEDs, d'un port USB... En résumé, ce mote est bien équipé. Certains motes possèdent même un connecteur SMA (SubMiniature version A) qui permet d'ajouter une antenne supplémentaire, dans le but d'améliorer la portée du noeud. A titre indicatif, le capteur de température détecte des températures allant de -40°C à 123°C, avec une précision de l'ordre de 0,4°C. Pour fonctionner, un mote a besoin de deux piles de 1,5V, sa durée de vie est donc limitée par le voltage des piles. En outre, il possède 10kB de mémoire vive et seulement 48kB de mémoire flash, ce qui est relativement peu. De plus, le microcontrôleur utilise des entiers codés sur 16 bits, il ne dispose pas d'une unité de calcul en virgule flottante, ce qui peut s'avérer gênant, nous en reparlerons.

Lors d'un déploiement, les motes sont répartis de manière à assurer la couverture complète d'une une zone prédéfinie. Afin de collecter les données captées par les noeuds ainsi dispersés, il est nécessaire de connecter un mote sur une machine disposant d'un port USB, on le nomme alors « sink » dans la mesure où il joue le rôle de « puit », c'est-à-dire qu'il est chargé de récupérer toutes les données envoyées par les autres noeuds du réseau. En réalité, ces derniers peuvent être relativement éloignés du sink, et ne peuvent donc pas envoyer directement au sink les données qu'ils viennent de capter. A noter que les données envoyées par les noeuds concernent aussi bien leurs capteurs, des informations relatives au réseau, le voltage des piles ainsi que d'autres informations

utiles. Dans le réseau de capteurs sans-fil que nous souhaitons déployer, chaque noeud communique avec ses voisins en Wi-Fi sur la bande de fréquence 2,4GHz.

C'est ici qu'intervient le protocole de routage RPL (Routing Protocol for Low power and Lossy Networks). Chaque paquet IPv6 du réseau doit passer par un chemin pour arriver jusqu'au sink. L'algorithme RPL construit un graphe orienté de noeud et permet ainsi la découverte d'une route entre un noeud émetteur et le sink. RPL assure ainsi une topologie dynamique du réseau, dans le cas où un noeud capteur est ajouté, déplacé, retiré, ou défaillant. Chaque mote doit donc être en mesure de relayer les paquets de ses voisins. Il doit alors pouvoir jouer à la fois le rôle d'émetteur et celui de récepteur.

Pour ce faire, chaque mote doit tout d'abord disposer d'un système d'exploitation. Nous utilisons le système d'exploitation libre ContikiOS. Étant conçu spécialement ce type de petits appareils en réseau, ContikiOS a l'avantage d'être léger et de favoriser une consommation énergétique minimale. De plus, il supporte les protocoles IPv6 et 6LoWPAN (IPv6 Low power Wireless Personal Area Network). Cela s'avère particulièrement utile dans la mesure où nos motes communiquent en IPv6 et utilisent le protocole de communication 802.15.4 défini par l'IEEE (Institute of Electrical and Electronics Engineers).

Pour que chaque noeud sache ce qu'il doit effectuer, nous avons « flashé » d'une part les motes « émetteurs » avec un programme écrit en langage C (udp-sender.c), et d'autre part le mote sink avec un programme C différent (udp-sink.c). ContikiOS se charge alors d'allouer les ressources processeur afin d'exécuter le programme présent sur le noeud capteur. Les motes ainsi flashés utilisent le protocole de transport UDP (User Datagram Protocol) au dessus du protocole IPv6 pour la couche réseau. Utiliser le protocole TCP (Transmission Control Protocol) n'aurait pas été envisageable ici, dans la mesure où un noeud destinataire aurait dû acquitter les segments reçus, et cela aurait entrainé une consommation énergétique plus importante. En revanche, des pertes de paquets peuvent avoir lieu, nous reviendrons sur ce point ultérieurement.

2.1.2 Logiciels existants

Après avoir collecté les données, nous voulons les exploiter. Le logiciel Collect-View répond à ce besoin, il est fourni avec Contiki. Il a été développé en Java et dispose d'une interface graphique réalisée en Swing. Pour rappel, nous avons un sink connecté au port USB d'une machine. Sur cette machine nous installons alors le logiciel Collect-View. En lisant le port USB, Collect-View récupère les données provenant du sink. Ces données sont « brutes » et nécessitent d'être traitées. En effet, le microcontrôleur d'un mote ne pouvant manipuler des nombres flottants, certaines données telles que la température ne sont pas directement exploitables. Collect-View se charge alors d'effectuer des calculs de conversion sur les données brutes. Il les affiche ensuite sur une interface graphique, où l'on voit apparaître des courbes qui retracent l'évolution des différentes données sur une période de temps.

Lors de ce projet, un autre outil nous a été utile, il s'agit du simulateur Cooja. Ce dernier est également fourni avec Contiki. Il permet de simuler un réseau de motes virtuels. Grâce à ce simulateur, nous pouvons ainsi tester rapidement un code écrit en langage C, sans avoir besoin de flasher de vrais motes. Nous pouvons répartir un nombre quelconque de noeuds sur une zone donnée. Nous visualisons alors en temps réel (ou accéléré) la topologie du réseau, ainsi que les routes empruntées lors des transitions de paquets. Ce simulateur se couple très bien avec Collect-View pour réaliser des mesures statistiques du réseau.

Le logiciel de collecte de données Collect-View et le simulateur Cooja sont deux bons outils à notre disposition. Cependant, ils sont insuffisants pour ce projet puisque nous voulons centraliser les données et les mettre à disposition. C'est pourquoi nous avons choisi de développer notre propre outil, qui sera détaillé par la suite.

2.2 Méthode et choix techniques

2.2.1 Planning

Pour parvenir à nos fins tout au long du projet, nous avons progressé par paliers. Nous nous sommes fixés des objectifs intermédiaires et les avons respectés. Nous organisions également des réunions régulières, environ toutes les deux semaines, avec nos collaborateurs et encadrants de projet afin de discuter de l'avancement, de faire des mises au point, ou encore de discuter des choix techniques et d'architecture du logiciel.

En ce qui concerne le travail fourni, nous faisions le nécessaire pour se voir régulièrement et travailler ensemble le plus possible. Nous estimons avoir consacré de 8 à 12h par semaine de travail en moyenne, comprenant recherche, lecture d'articles, auto formation sur certaines technologies, conception logiciel, programmation et tests.

2.2.2 Choix techniques

Au niveau applicatif et choix techniques, notre but était de centraliser les données émises par les capteurs tout en offrant une interface d'accès à celles-ci. Pour cela, nous avons décidé de mettre en place un SGBD relationnel. De plus, dans l'optique d'une accessibilité optimale, nous avons choisi de créer une application web accessible sur internet, ce qui permet la compatibilité avec pratiquement tous les terminaux possédant une connexion vers internet : ordinateurs, smartphones, tablettes...

Le lien entre le réseau de capteurs et l'application web est gérée par une passerelle programmée en Java. Celle-ci fait l'objet d'un paragraphe détaillé dans la partie suivante.

Nous avons choisi pour notre application côté serveur la technologie JEE (Java Enterprise Edition). JEE est une spécification de techniques axées autour du langage Java créée par l'entreprise Oracle et destiné principalement pour la création d'applications d'entreprise. On qualifie ces applications de multi-niveaux car elles possèdent dans la plupart des cas une partie cliente, une partie serveur d'application, et un serveur de données. Cette technologie possède un panel d'outils très riche et complet gravitant autour d'elle. En effet, nous utilisons certaines d'entre elles dans ce projet comme l'interface JPA (Java Persistence API) et son implémentation EclipseLink. Ceci permet une abstraction entre les données et la couche métier de notre application en fournissant des méthodes de persistance avec du JPQL (Java Persistence Query Language) et où les requêtes SQL (Structured Query Language) sont masquées au développeur, bien qu'il puisse tout de même en faire. La gestion de connexion SQL et déléguée à un pool de connexion BoneCP. Etant donné que nous avons une base de données de type MySQL, nous utilisons également le driver de connexion JDBC. Nous utilisons également dans la partie serveur la spécification EJB (Entreprise JavaBeans).

Dans l'optique de données ouvertes, nous avons créé une API de type REST que n'importe quel développeur peut interroger afin de produire une application future se basant sur ces données. Pour cela nous utilisons JAX-RS 2 (Java API for RESTful Web Services), qui nous facilite grandement la tâche qu'incombe la création d'une telle API.

Le serveur d'applications que nous avons décidé d'utiliser est Glassfish 4, avec la version Java EE 7 SDK. Il est open-source et contrairement à d'autres serveurs comme Tomcat, il intègre toutes les spécifications JEE que l'on utilise sans aucun besoin d'ajout externe. De plus, il intègre une interface web d'administration complète, ce qui est appréciable et facilite certaines actions comme le déploiement.

Concernant la partie cliente, nous utilisons les technologies JSP (Java Server Pages) et JSTL (Java Standard Tag Library). JSP permet l'insertion de code Java dans les pages web HTML afin d'afficher le contenu de variables par exemple. JSTL est une surcouche qui permet l'utilisation de balises proches du HTML pour effectuer des actions qui concernent du contenu back-end tout comme JSP. Cependant cela permet une séparation nette des vues et des parties serveur puisque aucun code Java n'est présent dans les vues. Cela pourrait par exemple permettre à un collaborateur ne maîtrisant pas le Java de créer des vues quand même. Le visuel de la partie cliente se

fait grâce aux technologies HTML5 et CSS, couplé à JavaScript et la librairie JQuery pour le dynamisme des pages. Nous utilisons le framework CSS font-end Zurb Foundation dans l'optique d'un visuel agréable et confortable pour l'utilisateur et la librairie HighCharts pour la production de graphiques.

L'architecture globale ainsi que les technologies utilisées peuvent être résumées par ce schéma :

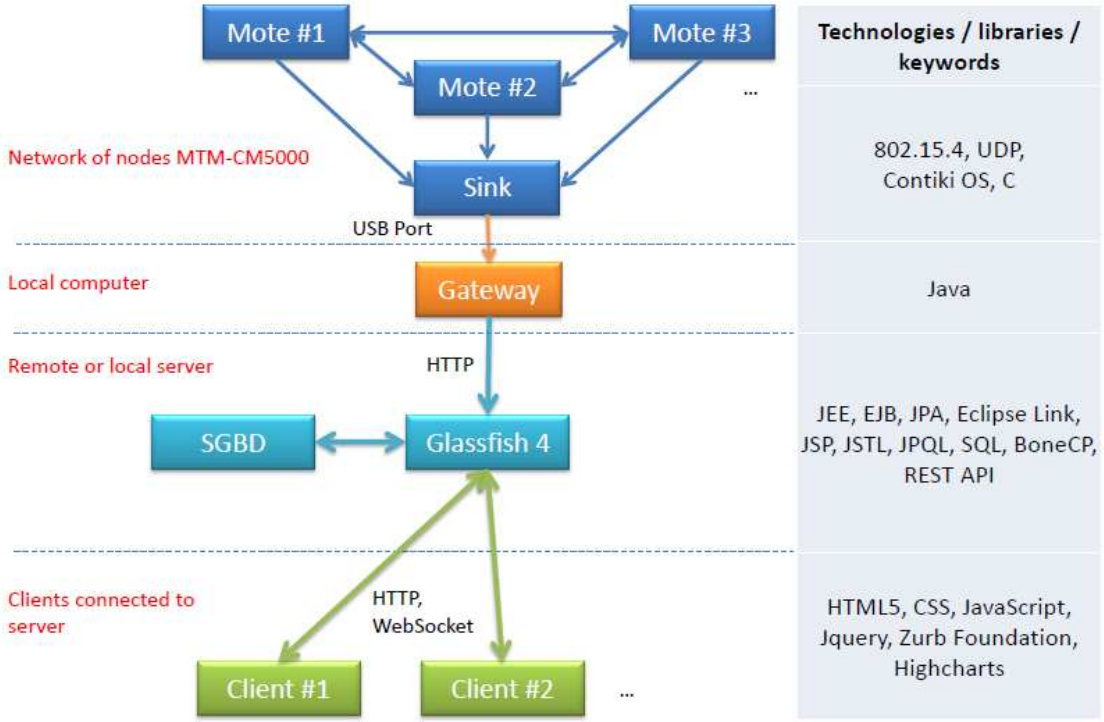


FIGURE 2.2: Flux de données entre le réseau de capteurs et l'application web

3 Solutions proposées

Pour répondre à nos objectifs et construire les différentes parties de l'application, nous avons utilisé les technologies précédemment présentées. Ces différentes parties de l'application seront décrites en détails dans cette section. Nous détaillerons dans un premier temps la passerelle, puis le modèle relationnel, et enfin l'application serveur.

3.1 La passerelle entre le réseau de capteurs et l'application serveur

Nous pouvons rappeler que les données du réseau de capteurs sont toutes envoyées à un capteur puit (sink mote), qui est connecté à un ordinateur via une liaison USB. Le rôle de la passerelle est de faire suivre les données du réseau vers l'application web.

Pour cela, nous avons mis en place une passerelle qui lit le port USB et ouvre une connexion TCP afin d'envoyer les données au serveur. Le protocole applicatif utilisé est HTTP, et une requête POST est réalisée sur l'API REST. Le serveur se charge ensuite de traiter les données et de les sauvegarder.

Cette passerelle est réalisée en Java et nous avons repris le code du modèle de l'application collect-view présente dans le système Contiki. Cette passerelle nécessite des fichiers compilés en C et Python (dossier tools) qui lui permettent de lire les données sur le port USB, récupérer la listes des capteurs du réseau, convertir les données reçues des microcontrôleurs MSP430. Ces fichiers sont aussi déjà présents dans le système Contiki. Ces fichiers sont compilés pour les systèmes Linux mais également Windows, ce qui permet à la passerelle d'être multiplateforme puisque le langage Java l'est.

Nous avons modifié le code original en ajoutant le transfert HTTP des données sur le serveur, mais également la prise en compte d'un nombre quelconque de données provenant des capteurs. En effet, initialement, le code présent vérifiait la présence de 30 entiers. Nous avons réduit le nombre d'entiers à ce qui nous intéresse dans le cadre du projet et des futurs travaux à 17 entiers. La passerelle fournit du feedback sur la connexion avec le serveur en affichant sa réponse. Lors de l'exécution, elle prend en paramètre l'URL sur laquelle les données vont être postées.

3.2 Le modèle relationnel

Le modèle relationnel de notre application comporte plusieurs entités.

Les expériences (ou mesures) : On associe les données à une expérience précise. Cette expérience comporte une description, des commentaires, et un booléen indique si elle est active ou non.

Les libellés : Ils représentent la sémantique de chaque information contenu dans le flux de données. Nous pouvons citer par exemple la température, l'humidité.

Les stratégies : Elles représentent le nom des différentes classes Java servant à la conversion des données brutes reçues par les capteurs. Par exemple la stratégie de conversion concernant la température est la suivante : -39.6 + 0.01 * température brute. Cela permet de la convertir en degré Celsius.

Les filtres : Ils permettent de lier une position d'entier contenu dans le flux de donnée avec une stratégie, et un libellé. Par exemple, si la température est en position 17, le filtre correspondant va lier cette position avec la stratégie température et le libellé température.

Les noeuds du réseau : Ils sont stockés avec une partie de leur adresse MAC/IPV6 codée sur 16 bits. On a également des champs prévus pour la position du noeud.

Les données : Les données sont représentées en nombre flottants, et sont liées à un noeud, un libellé, une expérience. La date et l'heure sont également enregistrés afin de la situer temporellement.

Les capteurs (sur les noeuds) : Ils peuvent mesurer différentes grandeurs (température, humidité, luminosité) et font l'objet d'une table dans la base de données, mais nous ne les utilisons pas dans ce projet. Cela permettrait d'associer les capteurs aux noeuds, d'indiquer s'ils sont soudés directement dessus, et dans le cas contraire, d'indiquer leur position.

Ces différentes entités permettent de gérer un nombre de données quelconque, avec chacune leur sémantique et leur stratégie, et tout cela sans toucher au code de l'application. Cela apporte donc de la flexibilité, ce qui est indispensable pour une vision à long terme.

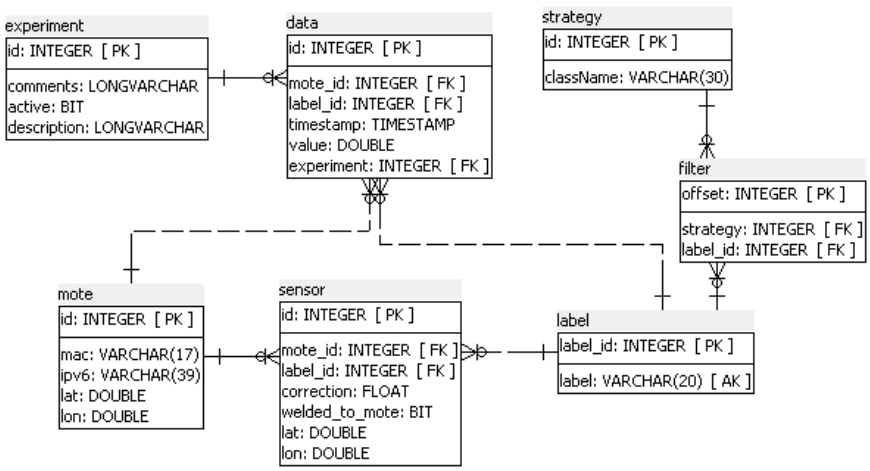


FIGURE 3.1: Modèle conceptuel de données

3.3 L'application serveur

3.3.1 Architecture globale

Les différentes parties de l'application

L'application a été construite selon une architecture modèle-vue-contrôleur (MVC). Le point d'entrée d'une requête HTTP est une servlet. Cette servlet fait office de contrôleur puisqu'elle contrôle la méthode HTTP employée, ainsi que les arguments présents, leur type et format.

Une fois les données contrôlées, et en fonction des arguments reçus, elle fait appel aux classes du modèle pour effectuer des transformations, des calculs, des sélections ou insertions dans la base de données... Dans le cas des données reçues du réseau de capteurs, leur traitement avant persistance est détaillé dans la section suivante.

Toutes les interactions avec la base de données passent par une couche d'abstraction construite selon le design pattern DAO (Data Access Object). Cette couche permet aux classes qui utilisent les objets métiers de s'abstraire totalement de la manière dont les objets sont persistés dans la base de données. Par ailleurs, ce modèle DAO que nous avons fait est une surcouche au dessus de l'implémentation de JPA qu'est EclipseLink, implémentation par défaut contenue dans le serveur d'applications Glassfish 4.

Une fois les données traitées, elles sont récupérées dans la servlet et envoyées à la vue qui va se charger de les afficher à sa convenance. Les vues sont des pages JSP qui incluent des tags JSTL comme présenté dans une précédente partie. Les formats de données des vues peut être variées. Par exemple, cela peut être du HTML, du JSON, des images... Les différents visuels exploitant les données des capteurs seront présentés dans une partie ultérieure.

Voici un schéma qui résume l'architecture de l'application serveur et les interactions entre les différents composants :

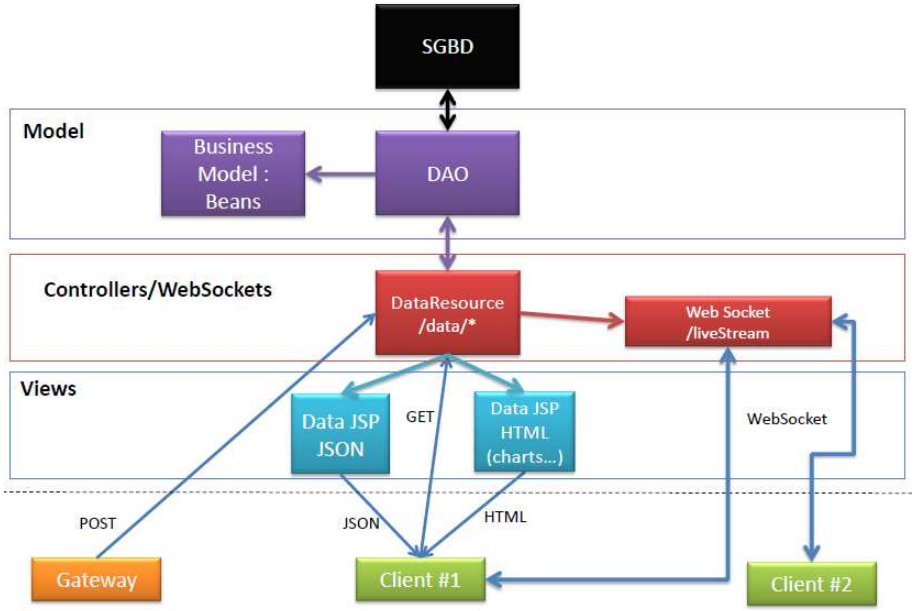


FIGURE 3.2: Architecture de l'application serveur et les flux de données entre les différents composants

Le standard JPA et les annotations EJB au sein de l'application

Le standard JPA impose l'utilisation d'annotations (annotations EJB) pour permettre de "mapper" les objets métiers (appelés entités Javabeans) et leur relations et permettre ainsi la communication avec la base de données d'une manière cohérente et transactionnelle. L'utilisation de ces annotations EJB est fondamentale pour permettre la gestion des beans. Un exemple d'annotations d'un bean est disponible en annexe. JPA permet l'insertion, la sélection, la suppression de données en faisant appel au gestionnaire d'entités (EntityManager) directement et donc sans manipuler de requêtes SQL. JPA confère au programme une abstraction sur le SGBD employé, et permet ainsi la compatibilité même après changement de SGBD, sous réserve d'utiliser les fonctions du gestionnaire d'entité et le langage JPQL et donc de ne pas utiliser de requête natives. Cependant, pour des requêtes plus compliquées que JPQL ne permet pas, il est tout à fait possible d'utiliser le langage SQL en faisant des requêtes natives sur le SGBD employé.

Les annotations EJB permettent de déléguer le cycle de vie de certains objets directement au conteneur de servlets. Cela permet d'améliorer les performances. En effet, ils sont instanciés juste après le déploiement de l'application et réutilisés à chaque requête contrairement à une instanciation

classique qui serait renouvelée à chaque requête. Par exemple, le gestionnaire d'entité, ainsi que tous les objets formant le DAO sont des objets sans état gérés par le conteneur.

3.3.2 Le traitement des données

Les données brutes provenant des noeuds du réseau et postées sur l'API sont récupérées dans la requête HTTP et traitées sur le serveur.

Tout d'abord, nous avons besoin de convertir les données reçues sous forme d'entiers dans des unités courantes. Par exemple nous convertissons la température en degré Celsius, l'humidité en pourcentage, ou encore la batterie restante en volts. Pour cela, nous avons créé une classe qui s'en occupe (RawData) et qui permet de récupérer les données converties sous forme d'objets de type Data, qui correspond à notre type personnalisé dans lequel la donnée est liée à son noeud, libellé, et expérience. Pour transformer les données, nous utilisons l'instanciation dynamique couplée au design pattern Strategy. L'instanciation dynamique nous permet d'instancier une classe dont le nom est récupéré depuis la base de données, et le design pattern Strategy nous permet de sélectionner un algorithme à la volée lors de l'exécution sous certaines conditions. Effectivement, les filtres récupérées depuis la base de données permettent d'instancier la bonne stratégie pour une position d'entier donnée. Les différentes stratégies ont la possibilité d'utiliser plusieurs entiers provenant du tableau de données brutes pour construire leur nombre flottant en retour.

Une fois les données converties, elles sont passées à l'objet DAO qui gère leur persistance afin de les enregistrer dans la base de données.

3.3.3 Mise à disposition des données

L'API REST

Nous avons décidé de mettre à disposition les données envoyées par les noeuds du réseau à disposition de tous à travers une API REST, ce qui permettrait à de futurs développeurs de créer des applications graphiques utilisant ces données.

L'API permet de récupérer toutes les données et informations souhaitées par le biais de filtres et renvoie des réponses de type JSON. Des statistiques peuvent être calculées par l'API comme des moyennes glissantes.

Tout d'abord, l'API permet de récupérer les données du modèle telles que les libellés, les stratégies, les filtres, mais également les motes présents dans le réseau. Il est aussi possible de modifier la position relative des noeuds, ce qui peut être utile pour les interfaces qui souhaitent placer géographiquement les noeuds sur une carte ou une image. Ces données peuvent être utilisées dans le futur pour une interface d'administration par exemple. De plus, les liens de l'API peuvent être protégés par une authentification.

Dans un second temps, l'API dispose de liens permettant d'ajouter des données concernant les noeuds, et de les récupérer. Ces données peuvent être par exemple la température, l'humidité, la luminosité...

Les filtres applicables concernent les libellés (température, humidité...), les noeuds (par leur adresse MAC), l'ID de l'expérience, et enfin la période de temps. Il est aussi nécessaire de préciser le nombre de données souhaité (par libellé et par noeud). Des moyennes sont calculées toutes les x valeurs pour satisfaire le nombre de données demandé.

L'URL de base de l'API actuelle est la suivante : http ://anthonyderoche.com :8081/pidr/rest
Voici ci-dessous les actions possibles avec les requêtes HTTP correspondantes :

Ajouter des données	POST /data
Récupérer des données en précisant les libellés	GET /data/{experiment_id}/{labels}/ {number_of_values}
Récupérer des données en précisant les libellés et les noeuds	GET /data/{experiment_id}/{labels}/ {number_of_values}/{motes}
Récupérer des données en précisant les libellés, les noeuds, et une période	GET /data/{experiment_id}/{labels}/ {number_of_values}/{motes} /{from_timestamp}/{to_timestamp}
Récupérer des données en précisant les libellés, et une période	GET /data/{experiment_id}/{labels}/ {number_of_values}/{from_timestamp} /{to_timestamp}
Récupérer la dernière donnée, par noeud et par libellé, en précisant les libellés	GET /data/{experiment_id}/{labels} /last
Récupérer la dernière donnée, par noeud et par libellé, en précisant les libellés et les noeuds	GET /data/{experiment_id}/{labels} /last/{motes}

Une documentation plus précise de l'API est disponible à cette adresse : http ://anthonyderoche.com :8081/pidr/api

On peut imaginer dans le futur une extension de cette API afin qu'elle fournisse d'autres services.

Un flux temps réel grâce au protocole WebSocket

WebSocket est un standard du web et un protocole de la couche application, apparu avec HTML5 standardisé par l'IETF (Internet Engineering Task Force) dans le RFC 6455 en 2011. Il vise à construire un canal bidirectionnel full-duplex entre un serveur web et un navigateur au dessus de TCP. Cela permet l'établissement d'une connexion permanente entre par exemple une application JavaScript et un serveur qui fournit un flux de données. Ce protocole est très bien adapté dans notre cas puisque les données reçues constituent un flux, et nous pouvons l'exploiter en temps réel sur les applications graphiques avec cette technologie.

Dans notre cas, c'est l'application JavaScript du navigateur qui initie le processus de connexion (handshake). Le serveur gère le protocole WebSocket grâce à l'implémentation intégrée dans Java EE 7. Lorsque le processus de conversion des données est terminé, elles sont transformées en JSON et poussées dans le canal de la WebSocket, ce qui permet à tous les navigateurs connectés dans les recevoir. L'application JavaScript se charge ensuite de les traiter et de les afficher.

La WebSocket actuelle s'inscrit bien dans l'optique des données ouvertes, puisqu'il est tout à fait possible de s'y connecter depuis n'importe quelle application, sous réserve que le cross-domain soit activé, c'est à dire le fait de pouvoir ouvrir une connexion depuis un domaine vers un autre.

3.3.4 Interfaces utilisateur

Nous avons fait plusieurs applications graphiques utilisant l'API et la WebSocket. Ces applications sont programmées en HTML5 avec les librairies JQuery pour faciler le développement en JavaScript et le framework CSS front-end Zurb Foundation pour un visuel agréable et confortable pour l'utilisateur.

Dans un premier temps, nous avons créé un programme en JavaScript se connectant à la WebSocket afin d'y récupérer les données en temps réel. Des graphiques sont construits au fur et à mesure que les données arrivent grâce à la librairie HighCharts. Nous avons des courbes représentant l'évolution de la température, de l'humidité, de la luminosité et enfin de la batterie restante, et cela pour chaque noeud du réseau. Le nombre de points affichés sur les courbes est réglable. La librairie HighCharts permet l'export des données affichées sur le graphiques en différents formats tels que le JPEG, le CSV, le PDF, ce qui peut s'avérer utile.
Le lien actuel de cette application est le suivant : http ://anthonyderoche.com :8081/pidr/live

Dans un second temps, nous avons créé une carte où l'on peut placer les noeuds sur un plan pour les situer géographiquement. Le plan peut être n'importe quelle image, mais nous avons mis dans notre cas le deuxième étage de TELECOM Nancy puisque nous avons fait un déploiement à cet endroit. Une partie ultérieure y est consacrée. Sur cette carte nous pouvons observer les différents noeuds colorés en fonction de leur température actuelle selon une échelle variant du bleu au rouge. De plus, des informations complémentaires sont affichées lors du passage de la souris sur un noeud, telles que la température, l'humidité et la luminosité et l'heure de réception de la dernière donnée.
Le lien actuel de cette application est le suivant : http ://anthonyderoche.com :8081/pidr/mapTN

Nous pouvons imaginer une multitude d'applications graphiques possibles exploitant les données fournies par l'API et la WebSocket. Celles actuellement développées sont incomplètes et feront probablement l'objet d'ajouts et de modifications futures.

4 Résultats des expériences et améliorations envisagées

4.1 Premières expériences

Au sein du LORIA

Avant de commencer ce projet, un de nos encadrants, Emmanuel NATAF, avait déjà réalisé le déploiement d'un réseau de capteurs au sein du LORIA. Son expérience s'est étalée sur plusieurs mois. Elle regroupait un ensemble de bureaux situés sur un même étage. Une vingtaine de capteurs était ainsi déployés, avec en moyenne deux capteurs par bureau. A ce moment, l'exploitation des données a été réalisée avec l'application Collect-View. L'expérience a montré un taux de perte de l'ordre d'un paquet sur deux. Ce chiffre important s'explique par le fait que les données envoyées par un noeud étaient au nombre de 30 lors de cette expérience, et ces dernières ne contenaient pas encore certaines informations utiles concernant le réseau. Par la suite, certaines données envoyées par un noeud ont été supprimées, d'autres ajoutées. Passant alors au nombre de 17, elles contiennent dorénavant de nouvelles informations nécessaires à la bonne communication du réseau. L'expérience a ensuite été réitérée, et affiche désormais un taux de perte de l'ordre de 5 à 10%.

Dans nos appartements respectifs

Le développement de tous les composants de notre architecture logicielle étant abouti, nous sommes en état de nous passer de l'outil Collect-View. Disposant de trois capteurs chacun, nous avons eu la possibilité d'effectuer deux expériences en parallèle. Elles ont eu lieu dans nos appartements respectifs sur une durée de quelques jours. Nos deux réseaux sont similaires, en ce sens que deux noeuds capteurs émettent leurs données vers un « mote sink ». Par l'intermédiaire de la même passerelle, nos deux sinks envoient les données collectées à l'application serveur.

Cette première expérience personnelle nous a permis de vérifier que, pour des réseaux différents, le déploiement parallèle de plusieurs capteurs de collecte n'est pas impossible. Le résultat de cette expérience est visible en annexe. La courbe retrace l'évolution des températures au fil d'une journée. A titre indicatif, les capteurs représentés en rouge et en vert sont situés dans le même appartement, tandis que les deux restants sont placés dans un autre. L'un des capteurs se situant près d'une fenêtre (le 77.106 représenté en noir), nous observons une belle variation de température, à savoir une augmentation durant la matinée, pour atteindre ensuite un pic de chaleur dans l'après-midi, et enfin redescendre à la fin de la journée.

4.2 Déploiement du réseau au sein de TELECOM Nancy

4.2.1 Présentation du déploiement

L'application fonctionnant, nous sommes en mesure de réaliser notre expérience à TELECOM Nancy. Après avoir informé tout le personnel de l'école, le déploiement de capteurs au sein de l'école pouvait commencer. Nous avons choisi de couvrir l'intégralité du deuxième étage, qui se compose principalement de bureaux pour les enseignants et le personnel administratif. Cette zone étant relativement grande, nous avions hâte d'étudier les performances d'un réseau de capteurs sans-fil disposant d'un unique sink.

Pour cette expérience, nous disposions de 39 motes au total dont 7 avec antenne, soit 38 noeuds capteur et un sink. Concernant notre démarche, nous avons commencé par flasher les capteurs (assez rapidement avec un hub USB). Nous avons ensuite étiqueté les noeuds qui ne l'étaient pas avec leur adresse MAC. Cette étape est indispensable pour connaître la position de chaque mote

une fois qu'ils seront placés. Dans un souci de positionner le sink au centre de l'étage, nous avons choisi de le connecter à une machine (qui nous était fournie) placée dans le bureau du directeur de l'école. L'étage regroupant environ 40 salles, nous avons décidé de répartir nos 38 noeuds dans 38 pièces différentes, soit un mote par salle.

Avec l'aide de nos encadrants, nous avons alors installé un à un les capteurs dans les différentes salles de l'étage. Pour chaque salle, nous avons en général placé le capteur en haut d'une armoire, ou bien sur le rebord d'une fenêtre, afin qu'il ne soit pas dérangeant pour les enseignants et le personnel administratif. Dès que nous posions un capteur, nous le reportions à la main sur le plan de l'étage, en notant son adresse MAC et son emplacement précis. Nous avons réparti les noeuds munis d'une antenne de la manière qui nous semblait la plus pertinente. Cela nous a permis ensuite de placer correctement les capteurs sur l'application web. La topologie du réseau est visible en annexe.

Par ailleurs, chaque noeud est configuré pour envoyer un paquet de données toutes les 5 minutes. En conséquence, après avoir déployé tous les capteurs, nous les avons placés au fur et à mesure sur l'interface graphique, par ordre d'arrivée des paquets. Mis à part quelques erreurs de numérotation sur certains noeuds ou des problèmes de piles usagées, tous les motes étaient opérationnels après une dizaine de minutes, même les plus éloignés (comme par exemple le mote 107.154 situé dans la salle du conseil). Le déploiement, qui s'est déroulé sur une matinée entière, est une réussite.

4.2.2 Résultats et analyse

Nous avons observé régulièrement le déroulement de l'expérience via l'interface web pendant 6 jours. Nous n'avons pas remarqué de dysfonctionnements sur cette période. Nous avons voulu faire des interprétations sur les données collectées. Concernant la température, nous avons observé que la partie nord du bâtiment est globalement plus froide que la partie sud.

Nous avons mesuré les taux de perte sur 6 jours et 7h, soit 543 600 secondes. Un paquet étant envoyée toutes les 300 secondes, il devrait théoriquement y avoir 1812 paquets envoyés pour chaque noeud. En comparant ce nombre avec le nombre de paquets réellement reçus par le sink, nous avons pu établir un tableau qui répertorie les taux de perte pour chaque noeud. Celui-ci est disponible en annexe. Globalement nous avons constaté qu'il y a une corrélation entre le taux de perte et l'éloignement du noeud par rapport au sink. Plus ce dernier est éloigné, plus le taux de perte est important. Cependant, ceci est variable selon la topologie du réseau. De plus, la présence d'une antenne a un impact sur le taux de perte. En effet, nous avons effectué les comparaisons suivantes :

1. Taux de perte pour 2 noeuds très éloignés du sink :
   - 10.5% pour celui avec antenne (noeud 32.139 dans le bureau de M. Badonnel)
   - 25.2% pour celui sans antenne (noeud 41.154 dans le bureau de M. Tomczak)
2. Taux de perte pour 2 noeuds sans antenne :
   - 4.9% pour celui proche du sink (noeud 141.56 dans le bureau de M. Festor)
   - 27.2% pour celui éloigné du sink (noeud 97.145 dans le bureau de M. Van-Phuc Do)
3. Taux de perte pour 2 noeuds proches du sink :
   - 4% pour celui avec antenne (noeud 200.124 dans la salle stages)
   - 3.8% pour celui sans antenne (noeud 25.141 dans la salle PAO)

D'après les comparaisons 1 et 3, nous constatons que la présence d'une antenne pour des noeuds éloignés permet de réduire fortement le taux de perte tandis que pour des noeuds proches du sink, sa présence est superflue. La comparaison 2 nous confirme que le taux de perte augmente avec l'éloignement du noeud par rapport au sink.

Afin de réduire les taux de perte des noeuds éloignés, nous pourrions envisager un réseau constitué de plusieurs capteurs de collecte (sink). Cependant cette solution est difficile à mettre en place à l'heure actuelle à cause des problèmes induits tels que la duplication des paquets reçus et l'établissement de la topologie du réseau.

Après quasiment une semaine de collecte, nous avons dépassé le million de lignes (tuples) enregistrées dans la base de données. Cela entraîne des ralentissements concernant les requêtes SQL, le traitement des données, et donc une augmentation du temps de réponse de l'API. A terme, nous pourrions envisager de passer à une base de données non relationnelle afin de mieux gérer la masse de données (big data).

5 Conclusion

A l'issue de ce projet, nous avons atteint nos objectifs qui étaient de déployer un réseau de capteurs sans-fil au sein de TELECOM Nancy, ainsi que concevoir et développer toute une architecture logicielle visant à la collecte des données provenant de ce réseau. Dans la tendance des Open Data, nous avons développé une API par le biais de laquelle nous mettons à disposition les données collectées aux potentiels développeurs, qui seraient ainsi libres de réaliser une application les exploitant.

L'application que nous avons développée nous a permis d'exploiter les données collectées lors de l'expérience à TELECOM Nancy. Après avoir analysé les résultats, nous avons été en mesure de mettre en évidence un certain nombre d'améliorations possibles, telles que le passage du modèle de données en non relationnel (NoSQL) ou encore la collecte de données à partir de plusieurs sinks.

Par ailleurs, nous avons accordé une grande importance à l'architecture de notre projet, c'est pourquoi nous avons mis en place un certain nombre de patrons de conception. Ces derniers ont rendu notre code propre et évolutif dans la mesure où nous avons pu facilement modifier certains points de notre projet sans que cela entraîne d'importants changements.

Ce projet nous a permis d'approfondir nos connaissances dans de nombreux domaines informatiques, à savoir le logiciel embarqué, le réseau, le génie logiciel ainsi que les bases de données. Nous avons eu la possibilité d'apprendre et de mettre en pratique les technologies autour des web services avec Java EE. Cette technologie étant très populaire en entreprise, cela rend ce projet encore plus enrichissant, et cela ne peut être que bénéfique pour notre future vie professionnelle.

6 Bibliographie / Webographie

Aspect mote et son système

- Documentation technique d'un mote : http ://www.advanticsys.com/shop/mtmcm5000msp-p-14.html
- Tutoriel d'installation de Contiki : http ://www.contiki-os.org/start.html
- Code source de Contiki : https ://github.com/contiki-os/contiki
- Code source de Collect-View : https ://github.com/contiki-os/contiki/tree/master/tools/collect-view
- Documentation 6LoWPAN : http ://fr.wikipedia.org/wiki/6LoWPAN

Développement

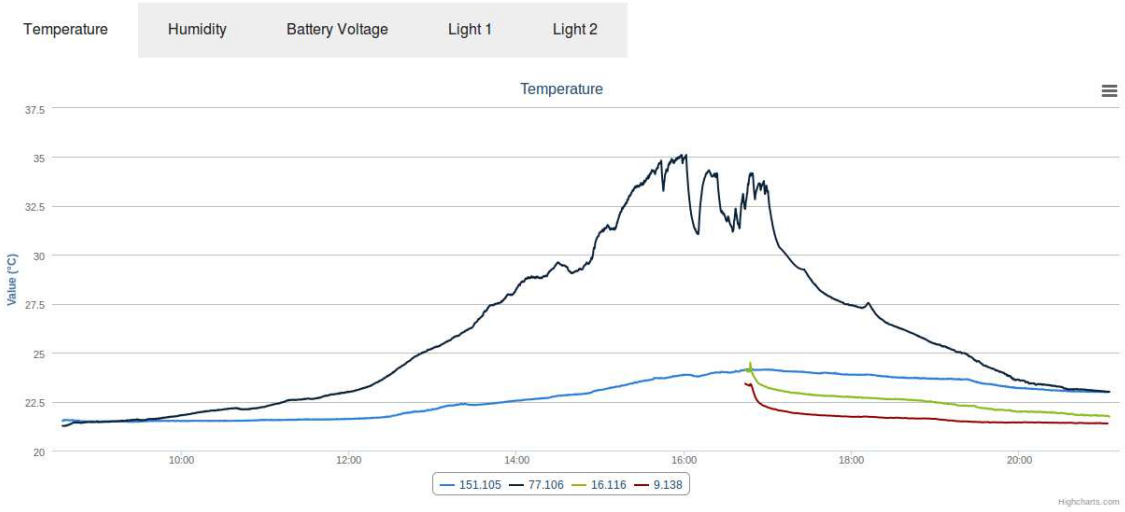
- Serveur web glassfish : https ://glassfish.java.net/
- Tutoriel JEE : http ://fr.openclassrooms.com/informatique/cours/creez-votre-application-web-avec-java-ee
- Tutoriel JAX-RS : http ://docs.oracle.com/javaee/6/tutorial/doc/giepu.html
- Tutoriel WebSocket Java : http ://docs.oracle.com/javaee/7/tutorial/doc/websocket.htm
- Documentation Zurb Foundation : http ://foundation.zurb.com/docs/
- Documentation JQuery : http ://jquery.com/
- Documentation HighCharts : http ://www.highcharts.com/

7 Annexes

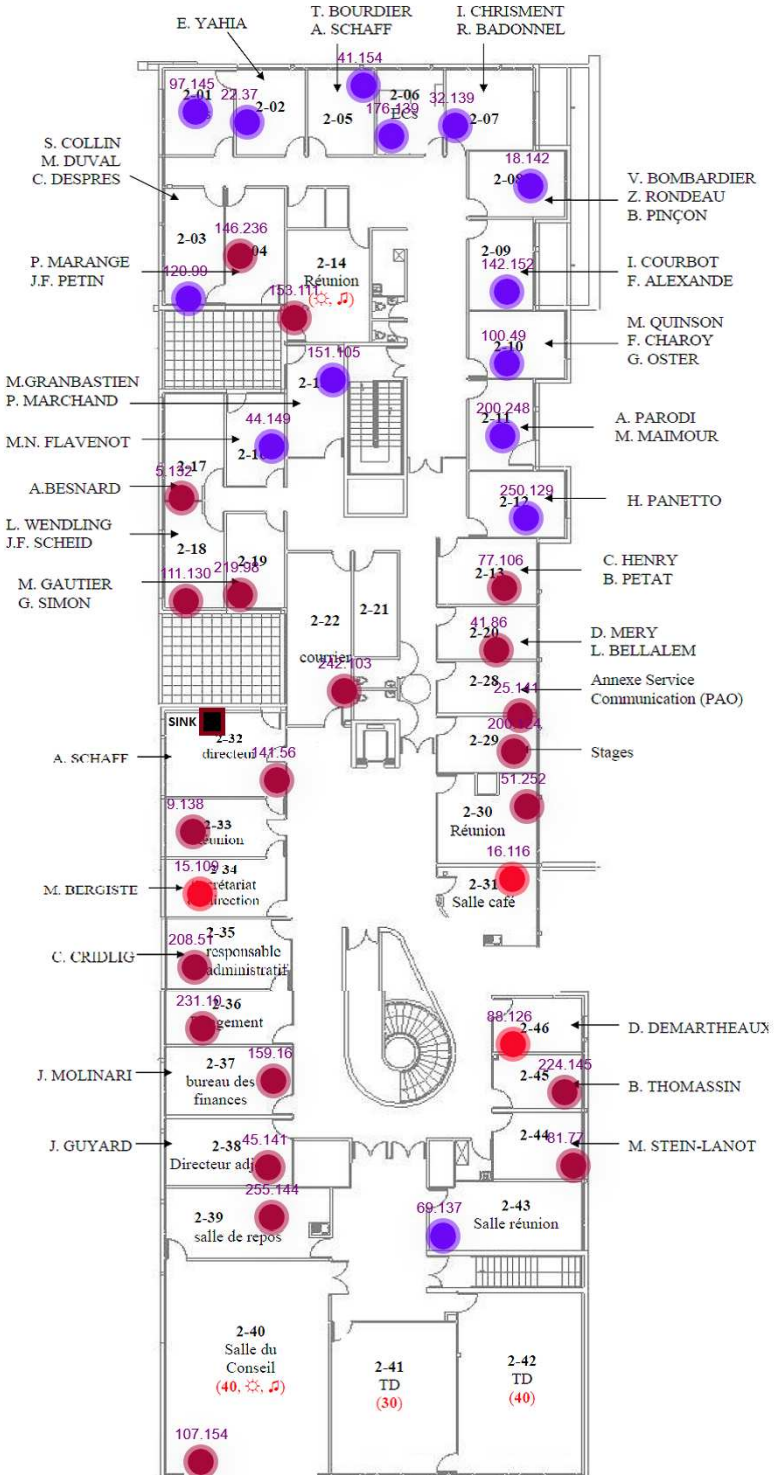
Exemple d'annotations EJB d'une entité

```
package pidr.beans.entity;

import java.sql.Timestamp;
import java.util.Locale;

import javax.json.Json;
import javax.json.JsonObject;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name = "Data")
public class Data implements JsonEncodable {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private int id;

	@ManyToOne
	@JoinColumn(name = "mote_id", referencedColumnName = "id")
	private Mote mote;

	private double value;

	@ManyToOne
	@JoinColumn(name = "label_id", referencedColumnName = "label_id")
	private Label label;

	private Timestamp timestamp;

	@ManyToOne
	@JoinColumn(name = "experiment", referencedColumnName = "id")
	private Experiment experiment;

	public Data(double value, Label label, Timestamp timestamp, Mote mote,
			Experiment experiment) {
		this.value = value;
		this.label = label;
		this.timestamp = timestamp;
		this.mote = mote;
		this.experiment = experiment;
	}

	// getters, setters...
}
```

Expérience dans nos appartements : évolution de la température au cours d'une journée



Plan du déploiement dans TELECOM Nancy



Taux de perte des noeuds du réseau déployés à TELECOM Nancy

Taux de perte (%)	Adresse du mote
3.3	16.116
3.4	9.138
3.5	231.10
3.6	45.141
3.6	88.126
3.7	51.252
3.7	242.103
3.8	25.141
3.8	208.51
4.0	15.109
4.0	200.124
4.4	250.129
4.7	159.16
4.9	141.56
4.9	41.86
5.1	219.98
6.0	200.248
6.1	77.106
6.4	111.130
7.7	142.152
8.0	151.105
8.9	81.77
10.5	107.154
10.5	32.139
11.6	44.149
12.0	100.49
12.1	5.132
12.3	69.137
15.0	146.236
15.7	153.111
18.0	224.145
18.5	18.142
19.0	120.99
23.2	22.37
25.2	41.154
26.1	176.139
27.2	97.145
39.5	255.144